A.1 Defining "Builtin" Modules for Maude Language Definitions

When defining language definition in Maude, for example using the operational semantics styles discussed in Chapter 3, one may naturally want to use the existing Maude data-types and libraries, particularly because they are well-engineered and thus can lead to efficient interpreters and tools for the defined languages. However, one needs to exercise care when importing Maude builtin modules, because it is very difficult, if not impossible, to reuse the names of the builtin operations as programming language constructs. The reason is that there is a very high chance that one wants to give those language constructs different attributes from those of the builtin operations, but Maude, for good reasons that go beyond our purpose in this book, does not allow it. For example, one may want to include an addition operation `_+_` in one's language, but one may not want it to be associative, commutative and have `0` as identity, like the `_+_` operation provided by the builtin Maude module `INT`.

In this section we give some Maude 2.4 specific hints on how one can appropriately redefine the Maude builtin modules to work smoothly with the Maude language definitions in Chapter 3. We advise the reader to run Maude without including its prelude, that is, with the command:

```
maude -no-prelude  <language-semantics>.maude
```

This way, no warnings will be issued when we redefine the builtin modules `INT` and `BOOL`. If for some reason one really needs to include the Maude prelude at start, then one should at least make sure that the `BOOL` module is not automatically included in other modules using the Maude command:

```
set include BOOL off .   --- not needed if one runs maude with flag -no-prelude
```

We can now define our own and safe `BOOL` module as follows:

```
mod BOOL is
  sort Bool .
  op true : -> Bool [ctor special (id-hook SystemTrue)] .
  op false : -> Bool [ctor special (id-hook SystemFalse)] .
  op if_then_else_fi : Bool Universal Universal -> Universal [poly (2 3 0)
    special (id-hook BranchSymbol  term-hook 1 (true)  term-hook 2 (false))] .
  op _==_ : Universal Universal -> Bool [poly (1 2) prec 51 special (
    id-hook EqualitySymbol  term-hook equalTerm (true)  term-hook notEqualTerm (false))] .
  op _=/=_ : Universal Universal -> Bool [poly (1 2) prec 51 special (
    id-hook EqualitySymbol  term-hook equalTerm (false)  term-hook notEqualTerm (true))] .
endm
```

The module above includes only a few components from the actual Maude builtin `BOOL` module. More precisely, it discards all the specific boolean operators for conjunction, negation, implication, etc., because one may want to define these as language constructs in one's language, possibly having a different semantics. For example, the conjunction `_and_` of IMP in Chapter 3 is short-circuited, so in particular it is *not* commutative (while the Maude `BOOL` builtin `_and_` is commutative). If one really needs to include the original Maude `BOOL` builtin operations, then we recommend that one change their names, like we do within the next module.

Like the module `BOOL` above, the module `INT` below defines only a portion of the Maude `INT` builtin module of integers. Unlike in `BOOL`, this time we also include some library operations on integers as part of the new builtin `INT`, but, however, we change their names by appending the word "`Int`" to their original names, e.g., `_+Int_`, `_<=Int_`, etc.:

```
mod INT is including BOOL .
  sort Int .
  op 0 : -> Int [ctor] .
  op s_ : Int -> Int [iter ctor special (id-hook SuccSymbol  term-hook zeroTerm (0))] .
  op -Int_ : Int -> Int [ctor special (id-hook MinusSymbol
    op-hook succSymbol (s_ : Int ~> Int)  op-hook minusSymbol (-Int_ : Int ~> Int))] .
  op _+Int_ : Int Int -> Int [assoc comm prec 33 special (id-hook ACU_NumberOpSymbol (+)
    op-hook succSymbol (s_ : Int ~> Int)  op-hook minusSymbol (-Int_ : Int ~> Int))] .
  op _/Int_ : Int Int -> Int [prec 31 gather (E e) special (id-hook NumberOpSymbol (quo)
    op-hook succSymbol (s_ : Int ~> Int))] .
  op _<=Int_ : Int Int -> Bool [prec 37 special (id-hook NumberOpSymbol (<=)
    op-hook succSymbol (s_ : Int ~> Int)  op-hook minusSymbol (-Int_ : Int ~> Int)
    term-hook trueTerm (true)  term-hook falseTerm (false))] .
endm
```

This way, one can still have full access to Maude's builtin libraries, but one elegantly avoids name clashes with homonymous operations that one may want to define as part of one's language syntax.

Finally, we define the `VAR` module. As discussed in Section 3.1.1, we want it to provide a sort `Var` and at least each letter in the alphabet as a constant of sort `Var` to be used in writing readable programs. We additionally include all the Maude builtin quoted identifiers (e.g., `'abc`, `'abc123`, etc.) as variables, so that one can use them as well:

```
mod VAR is
  sort Var .
  ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Var [format (g o)] .
  op <Qids> : -> Var [special (id-hook QuotedIdentifierSymbol)] .
endm
```